QAOOSE 2007

Proceedings of the 11th ECOOP Workshop on

Quantitative Approaches in Object-Oriented Software

July 31, 2007, TU Berlin, Germany

Fernando Brito e Abreu, Coral Calero,
Yann-Gaël Guéhéneuc, Christian Lange,
Michele Lanza, Houari A. Sahraoui,
Michael Cebulla (Eds.)

Bericht-Nr. 2007 – 4

ISSN 1436-9915

QAOOSE 2007

11th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering

July 31, 2007
Berlin, Germany

Workshop web site: http://www.inf.unisi.ch/lanza/QAOOSE2007/

Time	Speaker
09:00 - 09:15	**Opening**
09:15 - 10:00	Invited Talk by Horst Zuse, TU Berlin
10:00 - 10:30	**Coffee Break**
10:30 - 12:30	**Presentation Session**
	Inconsistencies of Metrics in C++ Standard Template Library? Zoltan Porkolab, Adam Sipos, and Norbert Pataki
	Automatic Generation of Strategies for Visual Anomaly Detection Salima Hassaine, Karim Dhambri, Houari Sahraoui, and Pierre Poulin
	Perception and Reality: What are Design Patterns Good For? Foutse Khomh and Yann-Gaël Guéhéneuc
	Session Discussion
12:30 - 13:30	**Lunch**
13:30 - 15:30	Discussions and/or Work on a common problem/project/paper
15:30 - 16:00	**Coffee Break**
13:30 - 15:30	Discussions and/or Work on a common problem/project/paper
19:00 - open	**Workshop Dinner**

Inconsistencies of Metrics in C++ Standard Template Library*

Zoltán Porkoláb, Ádám Sipos, and Norbert Pataki

Department of Programming Languages and Compilers, Eötvös Loránd University
Pázmány Péter sétány 1/C H-1117 Budapest, Hungary
{gsd, shp, patakino}@elte.hu

Abstract. Since McCabe's cyclometric measure, structural complexity have been playing an important role measuring the complexity of programs. Complexity metrics are used to achieve more maintainable code with the least bugs possible.
C++ Standard Template Library (STL) is the most popular library based on the generic programming paradigm. This paradigm allows implementation of algorithms and containers in an abstract way to ensure the configurability and collaboration of the abstract components. STL is widely used in industrial softwares because STL's appropriate application decreases the complexity of the code significantly.
Many new potential errors arise by the usage of the generic programming paradigm, including invalid iterators, notation of functors, etc.
In this paper we present many complexity inconsistencies in the application of STL that a precise metric must take into account, but the existing measures ignore the characteristics of STL.

1 Introduction

Structural complexity metrics play important role in modern software engineering. However, the software metrics are depend on used paradigm [10]. This fact makes hard to create multiparadigm metrics.

Generic programming is one the most untended paradigm from the view of paradigm, because most languages do not support this feature. C++ is multiparadigm language that suppport this paradigm [11]. Most important incarnation is the C++ Standard Template Library.

The C++ Standard Template Library (STL) is the most popular library based on the *generic programming paradigm* [1]. STL is widely-used, because the library is the part of the C++ Standard [11]. It consists of many useful generic data structures and generic algorithms, that work together with containers. STL is based on generalization and generalization results in simplified interface.

C++ STL consists of three main parts: containers, iterators and algorithms. Containers (e.g. vector, list, map, set, etc.) are the generalization of arrays, so they hold elements. Iterators guarantee access to the elements in containers.

* Supported by GVOP-3.2.2.-2004-07-0005/3.0

Iterators are nested types of containers. Iterators are a generalization of pointers, their standard interface originates from pointer-arithmetic. Algorithms are fairly irrespective of the used container, because they work with iterators. For instance, we can use the *for_ each* algorithm with all containers. The complexity of the library is greatly reduced because of this layout. As a result of this layout we can extend the library with new containers and algorithms simultaneously. This is a very important feature, because object-oriented libraries do not support this kind of extension. The C++ standard guarantees the complexity of the operations.

STL applies the generic programming paradigm, so we can expect that the common metrics can fail on this library because of the metrics' paradigm-dependence. As we will see, the old metric tools are not precise enough.

2 Positive effects

STL is a popular library, because it greatly reduces the complexity of a program from the view of programmers. The library offers many positive effects to code, but some of these effects cannot be measured by widely-used metrics.

STL makes the code more abstract, more powerful, more expressive, so programmers can avoid many mistakes [6]. STL is a standard library, many books and online references can be found (for example [1, 6, 11]).

3 Trivial inconsistencies

Many inconsistencies can be found between the common metrics and uage of STL. Some of these inconsistencies are quite clear.

One of the most obvious inconsistency is the widely-used object oriented metrics fail on C++ Standard Template Library, because this library is based on generic programming and implementing classes is unnecessary. Of course, we use objects and classes when the STL is applied, but we can write STL-based code without any new classes. Hence, the object-oriented metrics may fail on STL-based programs.

Another important feature is that STL is standardized library, so names of functions and classes in the library are well-known. The names express their behaviour, for instance the *copy* algorithm copies elements, the *sort* algorithm sorts a container, etc. No external library can achieve this important feature, and no existing metric can measure this special advantage.

STL has been designed as a generic programming library, so STL has a reduced interface: algorithms can be applied to more container types. The basic usage of the library is easy of attainment because of the reduced interface. This is a good feature, because beginner programmers do not shy away from STL. But this point is also not measured.

4 Complexity inconsistencies

In this section we examine some more sophisticated problems.

4.1 Error diagnostics

Error diagnostics usually do not matter when measuring software complexity. Metrics ignore syntactical and semantical errors in the code and usually examine programs as error-free software.

A simple mistake in STL-based code causes very long and incomprehensible error diagnostics. For example, more thousand character long error messages are not rare and often refer to unknown and unseen types and objects. Sometimes the errormessage points to the implementation of STL.

Some software tools help us to reduce the complexity of the messages, but these tools depend on the compiler and STL implementation.

Modification or maintain of STL-based code can be more difficult because of the complicate error diagnostics, so we should take it into account.

4.2 Functors

C++ functors are special objects that offer an *operator()* to simulate function-calls. Functors are quite common objects in STL-based code, because functors can avoid the overhead of non-inline functioncalls and some problems about the name of template functions to get the code to compile.

The problems of functors are their special requirements. Functor classes are often inherited from special classes that only support some typedefs. The names of these base classes are *unary_function* and *binary_function*. These base classes do not increase the complexity of a functor from the viewpoint of STL programmer.

Functors are always passed by value. Polymorphism and value passing an object do not work together, because the object would be sliced. So, polymorphic functors are not allowed.

4.3 Sorted ranges

Many problem arise from the inadequate usage of sorted ranges. Some algorithms have a special precondition, e.g. the input range must be sorted (for example, *binary_search*, *equal_range*, *set_union*, etc.). But the compilers do knot know what "sorted range" means, so the compiler cannot help us at this point. If we call an algorithm of this kind to an unsorted range, it causes undefined behavior. Unfortunately STLlint [12] cannot discover the improper usage of these algorithms. Using this kind of algorithms increases the complexity of the code.

Using the same sorting predicate to the sort and algorithm is important. If anyone violates this constraint it also leads to undefined behavior.

4.4 Dataflow

Dataflow models measure by the parameter-passing. This means the complexity of a program is based on parameters: how to read or write the arguments.

A basic problem is that we cannot read all parameterflows from an STL-based code. For example we write a functor and we call an algorithm with this functor as an argument. It is invisible that the code will execute the functor's functioncall operator.

Another problem is that we cannot decide if an algorithm modifies the container. For instance, let us consider the following two declarations. The find algorithm does not modify the container, but the sort algorithm does:

```
template <typename InputIter, typename T>
InputIter find(InputIter first, InputIter last, const T& t);

template <typename RanIter>
void sort(RanIter first, RanIter last);
```

On the other hand, the parameters are not independent. A container is passed by two iterators that define the range. If we call an algorithm usually call it with special iterators: begin and end iterators. It is so common that the programmers cannot make a mistake. So, iterators as parameters are very closely to count them twice.

4.5 Invalid iterators

Probably the most serious problem is usage of invalidated iterators. The compilers cannot help solve this kind of problem. Many kind of errors arise from usage of invalid iterators.

Different containers have different observance of iterator invalidating. The most trivial example of iterator invalidating is a reallocating vector, because their iterators do not point proper element of the given vector after a reallocating method.

This does not mean that the standard node-based containers are preferrel to contigous-memory containers. Both have advantages and disadvantages. C++ Programmers should know the rules of invaliding iterators.

5 Some proposals

In the paper [9] a multiparadigm metric is described. AV-graph measures three main points of a given program: the structure of the program, the dataflow in the program, and the complexity of the used data structures.

We have seen that the dataflow model is not precise enough. Informally speaking, the control structure also fails on STL-based code, because the usage of STL replaces many loops and if statemens.

It is also a common problem what can we mean by complexity of the STL's data structures. The complexity cannot be an STL implementation-specific value.

Complexity of STL's data structures should be based on some "semantical concepts": for instance, basic behaviour of the container (e.g. vector's reallocting strategy), special parameters of a data structure, how copying works, etc..

6 Conclusion

C++ Standard Template Library is a widely-used library based on the generic programming paradigm. Software metrics are mostly paradigm-dependent, so we can expect that the common metrics fail on C++ STL. In this paper we present many inconsistencies between STL and the widely used metrics. Our aim is to calibrate an old metric to measure STL-based code.

References

1. Austern, M. H.: Generic Programming and the STL. Addison-Wesley, 1999
2. Chidamber, S.R., Kemerer, C.F., A metrics suit for object oriented design. IEEE Trans. Software Engeneering, vol.20, pp.476-498, 1994
3. Fóthi Á. Nyéky-Gaizler J., Porkoláb Z.: The Structured Complexity of Object-Oriented Programs, Mathematical and Computer Modelling 38 pp.815-827., 2003
4. Howatt, J.W., Baker, A.L.: Rigorous Definition and Analysis of Program Complexity Measures: An Example Using Nesting, The Journal of Systems and Software 10, pp.139-150, 1989
5. McCabe, T.J., A Complexity Measure, IEEE Trans. Software Engineering, SE-2(4), pp. 308-320, 1976
6. Meyers, S.: Effective STL. Addison-Wesley (2001)
7. Pataki, N., Porkoláb, Z., Istenes, Z.: Towards Soundness Examination of the C++ Standard Template Library, In Proc. Electronic Computers and Informatics, ECI'06, Herl'any, 2006.
8. Piwowarski, R.E.: A Nesting Level Complexity Measure, ACM Sigplan Notices, 17(9), pp.44-50, 1982
9. Porkoláb, Z., Sillye, Á.: Towards a multiparadigm complexity measure, In. Proc of QAOOSE Workshop, ECOOP, Glasgow, pp.134-142, 2005
10. Seront, G., Lopez, M., Paulus, V., Habra, N.: On the Relationship between Cyclomatic Complexity and the Degree of Object Orientation, In Proc. of QAOOSE Workshop, ECOOP, Glasgow, pp. 109-117, 2005
11. Stroustrup, B.: The C++ Programming Language. Special Edition. Addison-Wesley, 2000
12. Gregor, D.: STLlint
    `http://www.cs.rpi.edu/~gregod/STLlint/`

Automatic Generation of Strategies for Visual Anomaly Detection

Salima Hassaine, Karim Dhambri, Houari Sahraoui, and Pierre Poulin

Dept. I.R.O., Université de Montréal

Abstract. An important subset of design anomalies is difficult to detect automatically in the code because of the required knowledge. Fortunately, software visualization offers an efficient and flexible tool to inspect software data searching for such anomalies. However, as maintainers typically do not have a background in visualization, they often must seek assistance from visualization expert. We propose an approach based on taxonomies of low-level analytic tasks, interactive tasks, and perceptual rules to design an assistant that helps analysts to effectively use a visualization tool to accomplish detection tasks.

1 Introduction

Although object-oriented programming has met great success in modeling and implementing complex software systems, practical experience with large projects has shown that programmers still face some difficulties with the maintenance of their code [10]. This is especially the case for design anomaly detection and correction. Design anomalies represent deviations from good object-oriented design that can hinder the maintenance and evolution of software projects. Anomaly detection is difficult to automate and may generate many false positives [9]. However, it can be greatly enhanced when combined with an appropriate form of visualization. Visualization offers powerful tools to foster a better understanding of software quality. It exploits the natural pattern recognition ability of the human brain and the knowledge of the expert to analyze data.

A number of visualization systems exist for complex software analysis [6, 8, 11, 15, 13]. However, each of these systems requires that the user explicitly specifies the visualization parameters. Building effective visualization also requires understanding some perception rules from cognitive sciences. Unfortunately, software maintainers typically do not have the necessary background in visualization, and therefore they often seek assistance from visualization experts to help them display their data and use efficiently the available tools.

This paper presents an approach which enables the user to specify a detection task in terms of software metrics and data analysis techniques, and then transforms it into a detection strategy with interactive visualization. This strategy assists the user in carrying on his task using a specific visualization tool. The approach also generates for each task a mapping between metrics and graphical representations based on perceptual rules.

The rest of the paper is structured as follows. Section 2 presents an overview of the approach and shows how its various components work together. Section 3 and Section 4 describe the two models involved in our approach. Section 5 details the transformation mechanisms between the two models. Section 6 illustrates with a concrete example how a general analysis task description can be transformed by our approach to be applicable for a specific visualization tool. Finally, Section 7 discusses some conclusions and identifies future work directions.

2 Overview

We propose a task-driven approach for software visualization in a way that supports user-defined analysis goals. We cover the entire process from the analysis task description to the generation of a sequence of interactive visualization actions supported by a specific software visualization tool. Figure 1 illustrates the method which consists of the following steps:

1. **Analysis Task (AT) Model**: This model offers a language to formally describes in terms of user goals the scenario for a detection task, based on a known analytic task taxonomy [1]. This model is independent from visualization.

2. **Interactive Visual Task (IVT) Model**: This model allows to describe the interactive tasks that a software visualization application should support.

3. **Analysis Task to Interactive Visual Task Transformation**: This mechanism transforms a detection task described using the Analysis Task Model into an interactive visual task conforming to the Interactive Visual Task Model. It is based on a set of perceptual rules.

4. **Transformation Session**: A transformation session is specific to a particular tool. It instantiates the above mechanism, taking into account the tool specification.

3 Analysis Task Model

This section presents the description of the Analysis Task Model. We start by presenting the basic operators that can be performed to explore the code (Section 3.1). Then we give the detection description model in terms of a goal-oriented modeling formalism (Section 3.2).

3.1 Taxonomy of Operators

In [1], Amar *et al.* proposes a low-level and domain-independent taxonomy of operators that a user might perform on a data set. Building on this taxonomy,

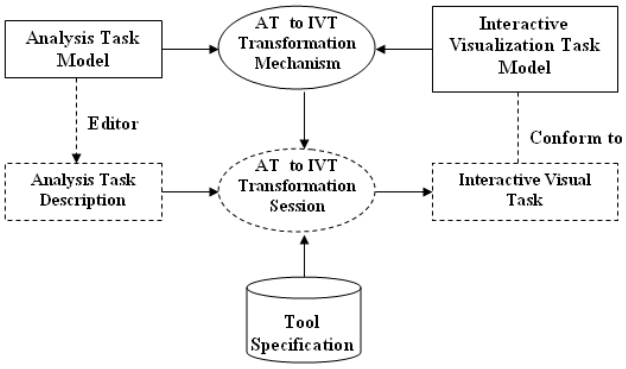

Fig. 1. Our proposed analysis-visualization pipeline.

we have derived a set of operators for the specific purpose of code-based data inspection. More specifically, we characterized each operator by the parameters required for its execution and the scope of its application. An operator is then described as a tuple $< operation, parameters >$ where *operation* is an action to perform on the data using the specified *parameters*. Parameters include code entity or set of entities (x or X), code metrics and relationships (Att or Atts), conditions (Cond), text labels (Label) and properties (Prop or Props) such as class name, class code, class position with respect to the package architecture, etc. Each operator has an area of effect: global vs. local. A global operator has to be applied to a large set of code entities (classes and interfaces), while a local operator has to be performed on a reduced subset of entities. Our set of operators is described in Table 1.

For example, when searching for classes that have extremely high coupling values globally in a program P, the operator to use would be *Find Extremum* with the parameters C, the set of classes in P, CBO as a coupling metric, and HIGH as the condition. Another example that involves a local operator is when a user needs to know the complexity of a particular class c. The appropriate operator is *Retrieve Value* with parameters c and WMC as a class complexity metric.

3.2 Modeling

Up-to-now, we have defined a set of basic operators that can be used in a detection task. The next step is to define a model that allows to describe a complete detection task. To this end, we defined a goal-driven model [14]. According to this model, a detection task consists of a goal, i.e., the purpose of the detection, that can be refined into sub-goals (see Figure 2). Each sub-goal is described by a list of steps. A step may be either one of the operators defined in Section 3.1 or a control structure (conditionals and iterators).

Operators	Scope	Description
Retrieve Value$\langle x, Atts\rangle$	local	Find the values of attributes $Atts$ for a code entity x.
Filter Attributes$\langle X, Atts, Conds\rangle$	global	Determine the subset of entity set X that satisfies the conditions $Conds$ on attributes $Atts$.
Filter Relationship$\langle X, Att, y\rangle$	global	Determine the subset of entity set X that are in relation Att with the entity y.
Find Extremum$\langle X, Att, Cond\rangle$	global	Find code entities in X possessing an extremely high or low value (specified by condition $Cond$) with regard to the value distribution of attribute Att.
Sort$\langle X, Att\rangle$	global	Rank code entities in X according to a given attribute Att.
Determine Range$\langle X, Att\rangle$	global	Determine the span of values of an attribute Att for a set of entities X.
Characterize Distribution$\langle X, Att\rangle$	global	Characterize the distribution of the values of attribute Att over a set of code entities X.
Cluster$\langle X, Atts\rangle$	global	Find clusters of similar values for attributes $Atts$ for a set of code entities X.
Correlate$\langle X, Att1, Att2\rangle$	local	Determine relationships between the values of two attributes $Att1$ and $Att2$ for a set of code entities X.
Verify Value $\langle x, Att, Cond\rangle$	local	Verify whether a condition $Cond$ is true for an attribute Att of an entity X.
Verify Property $\langle X, Prop, Cond\rangle$	global	Verify whether a condition $Cond$ is true for a property $Prop$ of a set of entities X.
Inspect$\langle x, Props\rangle$	local	Obtain detailed information about properties $Props$ for an entity x.
Save$\langle X, Label\rangle$	local	Attach the label $Label$ to the set of entities X.

Table 1. Analysis operators.

A detailed example of a detection task expressed using our model is given in Section 6.

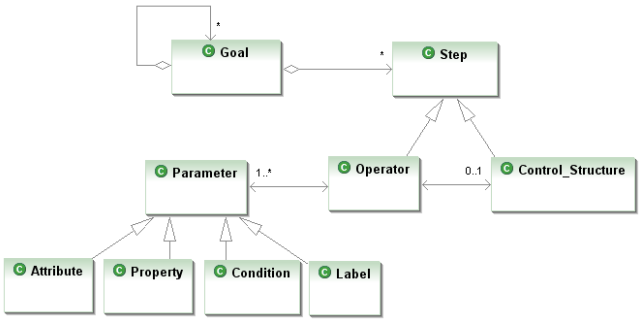

Fig. 2. Task Model.

4 Interactive Visual Task Model

This model is based on the *Task by Data Type Taxonomy* proposed by Shneiderman [12]. This taxonomy presents seven high-level interactive tasks that an information visualization application should support, such as *overview, zoom,*

filter, details-on-demand, relate, etc. These interactive tasks are task-domain information actions that users might want to perform in a visual environment.

In a visual environment, code entities are displayed as graphical representations. Properties of these entities are mapped onto the visual attributes of their representation. The visual attributes of a representation may change according to the interactive task performed on it. The various interactive tasks (call them interactors) are described as :

- **Overview**: Gain an overview of the entire collection of code entities.
- **Zoom**: Zoom in on entities of interest.
- **Filter**: Put emphasis on a sub-collection having certain properties, and filter out uninteresting entities.
- **Details-on-demand**: Select a particular entity and get details when needed.
- **Relate**: View relationships among code entities such as association, generalization, aggregation, etc.
- **History**: Memorize semantic information about a code entity, such as the role it plays in a design.
- **Selection**: Select certain entities of the collection.
- **Navigation**: Moving around within the collection.
- **Statistic Function**: Display the value of a function computed from entity attributes.

5 AT to IVT Transformation

The mapping process is a critical and challenging part of the visualization process. It has an important influence on the user interpretation of data. The mapping process consists of two phases. The first phase maps each analysis task operator to one or many interactive task interactors conforming to the Interactive Visual Task Model. The second phase maps the attributes of the code entities to the visual properties of their graphical representations. Our transformation mechanism uses perceptual guidelines to build more effective visual mappings.

For a given transformation session, the user provides an analysis task description, which is transformed into a description of an interactive visual task for a specific tool. In this paper, we illustrate the mapping process using the visualization tool VERSO [5].

5.1 Task Mapping

For a tool to be compatible with our approach, it must support every interactor defined in Section 4. Table 2 presents the mapping between operators, interactors, and the corresponding operations in VERSO.

As an example, when we apply the operator 'Find Extremum$\langle X, Att, Cond\rangle$', the first interactor is 'Overview', in order to get a global view of the data set X. In VERSO this interactor is implemented as the 'Zoom-out' operation. Then a

'Statistic Function' is applied to visualize the distribution of values for attribute *Att*. In VERSO, that interactor corresponds to the statistic filter. Finally, the interactor 'Selection' is applied to identify classes having an extreme value for the attribute, either high or low (depending on the condition *Cond*).

Operators	Location	Interactors	VERSO Operations
Retrieve Value$\langle x, Atts\rangle$	local	Zoom Details-on-demand	Zoom-in Display metric list.
Filter Attributes$\langle X, Atts, Conds\rangle$	global	Overview Filter Selection	Zoom-out Select feature
Filter Relationship$\langle X, Att, y\rangle$	global	Overview Relate Selection	Zoom-out Relationship filter Select feature
Find Extremum$\langle X, Att, Cond\rangle$	global	Overview Statistic Function Selection	Zoom-out Statistic filter Select feature
Sort$\langle X, Att\rangle$	global	Overview Navigation	Zoom-out Iterator
Determine Range$\langle X, Att\rangle$	global	Overview Statistic Function	Zoom-out Statistic filter
Characterize Distribution$\langle X, Att\rangle$	global	Overview Statistic Function	Zoom-out Statistic filter
Cluster$\langle X, Atts\rangle$	global	Overview Filter Selection	Zoom-out Select feature
Correlate$\langle X, Att1, Att2\rangle$	local	Zoom Statistic Function	Zoom-in Statistic filter
Verify Value $\langle x, Att, Cond\rangle$	local	Zoom Details-on-demand	Zoom-in Display metric list
Verify Property $\langle X, Prop, Cond\rangle$	global	Overview	Zoom-out
Inspect$\langle x, Props\rangle$	local	Zoom Details-on-demand	Zoom-in Display source code
Save$\langle X, Label\rangle$	local	Zoom Selection History	Zoom-in Select feature Tag feature

Table 2. Task Mapping.

5.2 Attribute Mapping

The attribute mapping provides support for the user by mapping data onto graphical primitives. As several mappings are possible, selecting the most appropriate mapping among all alternatives for a given situation usually requires considerable knowledge of visual perception principles, and of the data itself.

Through literature surveys [7, 4], we gathered several heuristic rules. These rules relate to the expressiveness and effectiveness of visualization primitives, such as color, size, etc. Expressiveness rules identify visualization primitives able of expressing the desired information, whereas effectiveness rules identify the most effective primitives for exploiting the capabilities of the output medium and the human visual system. Table 3 presents an example of the effectiveness rules proposed by Mackinlay [7].

In VERSO, each data entity is represented by a 3D box. Each representation has five visual attributes: color, height, twist, analog clock texture, and window texture. Figure 3 illustrates these representations.

Visual feature	Quantitative	Ordinal	Nominal
Position	0	0	0
Length	1	7	8
Angle	2	8	9
Slope	3	9	10
Area	4	10	11
Volume	5	11	12
Density	6	1	5
Color saturation	7	2	6
Color Hue	8	3	1
Texture	N	4	2
Connection	N	5	3
Containment	N	6	4
Shape	N	N	7

Table 3. Ranking of visual features. Features without a cost are not relevant to the corresponding measurement scale.

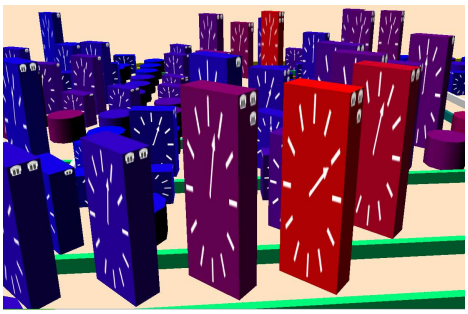

Fig. 3. Data entity representations in VERSO.

We can formalize the problem of generating a mapping between data attributes and visual features as a *valued constraint satisfaction problem*. Formally speaking, we define a 3-tuple $\langle X, D, C \rangle$ where:

- $X = \{X_1, X_2, ..., X_n\}$ is a finite set of variables.
- $D = \{D_1, D_2, ..., D_n\}$ is a collection of the domains of the variables in X, such that each variable X_i has a domain D_i, which is the set of possible values.
- C is a finite set of soft constraints. A soft constraint $f \in C$ is a function f_S on a set of variables $S \subseteq X$. S is the scope of the constraint.

For our problem, the variables are the data set to be visualized, and their domains contain the whole set of available visual features.

Function $f_S(\mathit{variable}, \mathit{operator}, \mathit{visualfeature})$, which is composed of soft constraints, evaluates the effectiveness of a visual feature for a given variable (*i.e.*, if a visual feature is the most effective for a given variable, the valuation takes the minimum value). These functions are based on perceptual rules and the scope (*local* or *global*) of the operator (see Section 3.1) that is parameterized by this variable. For example, texture should be used for local operators such as *retrieve*

value, because it is more effective when zooming in, but reduces its effectiveness during overviews.

Thus, the global cost function is the cost of a complete assignment; it is the sum of the costs expressed by each soft constraint f_S. A solution to our *valued CSP* is a complete assignment that has the minimal cost.

6 Case Study

In this section, we present a concrete example of a software analysis task using our approach. The task modeled is the detection of the *Blob*, a well-known anti-pattern described in [2]. The *Blob* is found in designs where one large class monopolizes the behavior and the other classes primarily encapsulate data. It is characterized by a class diagram composed of a single complex and non-cohesive controller class associated to simple data classes.

6.1 Analysis Task Description

The first step is to express the detection of the *Blob* as a sequence of actions to accomplish, as described by the Analysis Task Model (see Section 3). The specification for the *Blob* detection task is presented in Table 4. This description is based on software metrics and general data analysis techniques.

Method to accomplish Goal: *Blob* Detection
1: Find Extremum($wholeSystem$, WMC, WMC = HIGH)
2: Filter Attributes($extremeClasses$, LCOM5, LCOM5 = HIGH)
3: FOR EACH($c \in filteredClasses$) REPEAT(Step 4 to Step 6)
4: IF(Verify Value(c, DIT, $DIT = LOW$)) GOTO(Step 5) ELSE(continue)
5: Inspect(c, class name, method signatures)
6: IF(Verify Property(c, type, controller class)) GOTO(Step 7) ELSE(CONTINUE)
7: Save(c, controller class)
8: FOR EACH ($m \in controllerClasses$) REPEAT(Step 9)
9: **Accomplish Sub-Goal:** Data Class Verification(m)

Method to accomplish Goal: Data Class Verification($class$)
1: Filter Relationship($wholeSystem$, Association Relationship, $class$)
2: Filter Attributes($associatedClasses$,WMC = LOW, LCOM5 = LOW, DIT = LOW)
3: IF(Verify Property($filteredClasses$, type, data class))
4: Save($class$, *Blob*)

Table 4. *Blob* detection expressed using the AT model.

6.2 Generation of Interactive Visual Task

Once the *Blob* detection has been specified using the Analysis Task Model, a VERSO-specific interactive visual task is generated. Table 5 presents the attribute mapping and the task mapping.

Attribute Mapping
DIT ↔ number of windows
WMC ↔ height
LCOM5 ↔ twist

Method to accomplish Goal: *Blob* Detection
1: Zoom-out
   Apply the statistic filter for WMC on the whole system.
   Select classes having an extremely high WMC value.
2: Deselect classes that do not have a high LCOM5 value.
3: Iterate on the selected classes, for each class c REPEAT(Step 4 to Step 6)
4: Zoom-in.
   Display metric list of c.
   IF(c has a low DIT value) GOTO(Step 5) ELSE(continue)
5: Display source code of c.
   Inspect class name and method signatures of c.
6: IF(c is of type controller class) GOTO(Step 7) ELSE(continue)
7: Tag c as controller class.
8: Iterate on the controller classes, for each class m REPEAT(Step 9)
9: **Accomplish Sub-Goal:** Data Class Verification(m)

Method to accomplish Goal: Data Class Verification($class$)
1: Zoom-out.
   Apply the association filter for $class$ on the whole system.
   Select classes associated to $class$.
2: Deselect classes that do not have a low WMC value.
   Deselect classes that do not have a low LCOM5 value.
   Deselect classes that do not have a low DIT value.
3: IF($filteredClasses$ are of type data class))
4: Tag $class$ as *Blob*

Table 5. *Blob* detection expressed using the IVT model.

6.3 Application of the Interactive Visual Task

Figure 4 shows an example of a *Blob* detected using the interactive visual task guideline generated. The image on the left displays the result of the statistic filter on WMC (corresponding to Step 1 of the *Blob* detection method). The circled class was selected as a controller class. In the image on the right, the association filter is applied on that same controller class. Classes that kept their original color are associated with the controller class. Many of these classes are small, straight, and have few windows. Upon inspection of the code, we confirmed that we found a *Blob* occurrence.

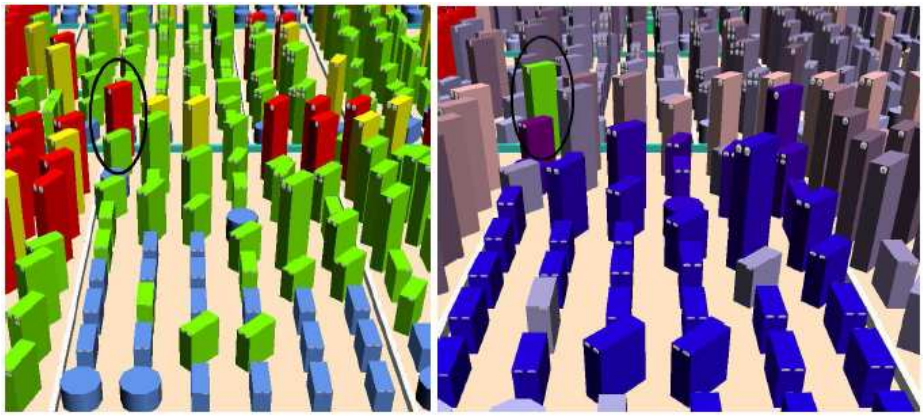

Fig. 4. Example of a *Blob* found in *PCGEN*. (Left) Box plot filter on WMC, the circled class is identified as a controller class. (Right) Association filter applied on the controller class. Inspection of the associated data classes confirms it is a *Blob*.

7 Conclusion

A lot of work has been done on the subject of analysis task taxonomies and interactive task taxonomies. For example, Wehrend and Lewis [16] proposed a taxonomy of cognitive tasks, such as identify, locate, distinguish, etc. This taxonomy has been extended by Zhou and Feiner [17] to automatically create multimedia presentations with their tool. Shneiderman [12] proposed a taxonomy of information-seeking visualization tasks. This work has been used in a software visualization context by Marcus *et al.* [8].

Concerning perception rules, Mackinlay [7] defined effectiveness and expressiveness criteria for graphical languages. Hikmet *et al.* [4] extended these rules for their tool Vista, using several studies on graphics, data visualization, visual perception, and psychology. Healey *et al.* [3] proposed an automated visualization assistant to help users construct perceptually optimal visualization, also relying on perceptual rules.

However, to our best knowledge, existing methods fail to combine analysis task taxonomies, interactive task taxonomies, and perceptual rules to create efficient mapping and visualization techniques that support the needs of the user.

In this paper, we proposed an approach to transform an analysis task description into an interactive visual task for a specific tool, using perceptual rules to choose the most appropriate data mapping.

As future work, we plan to incorporate a taxonomy for data models, which will allow us to extend our approach to be applicable on a larger set of visualization tools. Also, we plan to extend our knowledge base of perceptual rules in order to improve the attribute mapping generated by our approach.

References

1. Robert A. Amar, James Eagan, and John T. Stasko. Low-level components of analytic activity in information visualization. In *INFOVIS*, page 15, 2005.
2. William J. Brown, Raphael C. Malveau, III Hays W. McCormick, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley Press, 1998.
3. Christopher G. Healey, Robert St.Amant, and Mahmoud S. Elhaddad. Via: A perceptual visualization assistant. In *28th Workshop on Advanced Imagery Pattern Recognition (AIPR-99), Washington, DC.*, 1999.
4. Eve Ignatius Hikmet Senay. State of the art in scientific visualization. In *Siggraph 90*, 1990. Tutorial Notes.
5. Guillaume Langelier, Houari A. Sahraoui, and Pierre Poulin. Visualization-based analysis of quality for large-scale software systems. In *ASE*, pages 214–223, 2005.
6. Gerard Lommerse, Freek Nossin, Lucian Voinea, and Alexandru Telea. The visual code navigator: An interactive toolset for source code investigation. In *Proc. IEEE Symposium on Information Visualization*, pages 25–32, Washington, DC, USA, 2005. IEEE Computer Society.
7. Jock Mackinlay. Automating the design of graphical presentations of relational information. *ACM Trans. Graph.*, 5(2):110–141, April 1986.
8. Andrian Marcus, Louis Feng, and Jonathan I. Maletic. 3d representations for software visualization. In *SOFTVIS*, pages 27–36, 207–208, 2003.
9. Naouel Moha, Yann-Gaël Guéhéneuc, and Pierre Leduc. Automatic generation of detection algorithms for design defects. In *Proc. IEEE/ACM International Conference on Automated Software Engineering*, pages 297–300, September 2006.
10. Thomas M. Pigoski. *Practical Software Maintenance: Best Practices for Managing Your Software Investment*. Wiley, New York, 1996.
11. Steve P. Reiss and Manos Renieris. Chapter 11: The bloom software visualization system. In Kang Zhang, editor, *Software Visualization: From Theory to Practice*, pages 311–358. Kluwer Academic Publishers, 2003.
12. Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. *IEEE Visual Languages (UMCP-CSD CS-TR-3665)*, pages 336–343, 1996.
13. Michele Lanza Stéphane Ducasse. The class blueprint: Visually supporting the understanding of classes. *IEEE Trans. Softw. Eng.*, 31(1):75–90, 2005.
14. Allen Newell Stuart K.Card, Thomas P.Moran. *The psychology of human-computer interaction*. Lawrence Erlbaum Associates, 1983.

15. Maurice Termeer, Christian F. J. Lange, Alexandru Telea, and Michel R. V. Chaudron. Visual exploration of combined architectural and metric information. In *VISSOFT*, pages 21–26, 2005.
16. Stephen Wehrend and Clayton Lewis. A problem-oriented classification of visualization techniques. In *VIS '90: Proceedings of the 1st conference on Visualization '90*, pages 139–143, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.
17. Michelle X. Zhou and Steven K. Feiner. Visual task characterization for automated visual discourse synthesis. In *CHI '98: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 392–399, New York, NY, USA, 1998. ACM Press/Addison-Wesley Publishing Co.

Perception and Reality: What are Design Patterns Good For?

Foutse Khomh
Yann-Gaël Guéhéneuc
Ptidej Team, GEODES, DIRO, University of Montreal, Canada
{foutsekh,guehene}@iro.umontreal.ca

Abstract

We present a study of the impact of design patterns on quality attributes. An empirical study is performed by asking respondents their evaluations of the impact of all design patterns on several quality attributes. We present detailed results for three design patterns (Abstract Factory, Composite, and Flyweight) and three quality attributes (reusability, understandability, and expendability). We perform a Null hypothesis test and we conclude that, contrary to popular beliefs, design patterns do not always improve reusability and understandability, but that they do improve expandability.

1 Introduction

Many studies in the literature present design patterns as a promising solutions to improve the quality of object oriented software systems during development. It is widely claimed that they improve the quality of systems and that every well-structured object oriented architectures contain patterns [3].

However some studies suggested that the use of design patterns do not always results in good quality design. In particular, a tangled implementation of these patterns in a design impacts negatively the quality that these patterns claimed to improve [4]. Also design patterns generally increase the complexity of an initial design to ease future enhancements.

Thus, to the best of our knowledge, evidence of quality improvements through the use of design patterns consists primarily of intuitive statements and examples. There is little empirical evidence to support the claims of improved flexibility, reusability, adaptability as put forward in [3] when applying design patterns. Also, the impact of design patterns on other quality attributes is unclear.

This lack of evidence around the benefits of design patterns and their impact on design quality led us to carry out an empirical study on the impact of design patterns on the quality of systems as perceived by software developers and with respect to known principles of the object oriented paradigm.

In this work, we present a survey carried out over a population of experienced object-oriented developers and its results to attempt answering the question: *is the impact of design patterns on quality attributes positive, neutral, or negative?* We conclude by a discussion on the results.

2 Related Work

Since the introduction of design patterns by Gamma et al. [3], there has been a growing interest on the use of design patterns, many work have been carried out to study the potential impacts of this concept on software systems but very few investigated empirically the impact on quality. We present here only examples of the main work on design patterns.

Wydaeghe et al. [6] presented a study on the concrete use of six design patterns when building an OMT editor. They discussed the impact of these patterns on quality attributes such as reusability, modularity, flexibility, and understandability. They also discuss the difficulty of the concrete implementation of these patterns. They concluded that although design patterns offer a lot of advantages, not all patterns have the same effects on the quality attributes. However, this study is limited to the authors' own experience and thus their appreciation of the impact of these patterns on quality can hardly been generalized to any context of development.

Tahvildari et al. [5] studied the 23 design patterns from [3] and presented a layered classification of the primary relationships between these patterns: use, refine, and conflict, and three secondary relationships: similar, combine, and require (that can be expressed in terms of the primary ones). They organized the design patterns into two abstraction levels. They dis-

cussed how their classification can assist software engineers with understanding better the complex relationships between patterns, organizing existing patterns as well as categorizing and describing new patterns and building tools that support the application of patterns during restructuring. However, they did not investigate whether the use of these patterns really improve the quality of designs.

McNatt and Bieman [4] examined the coupling between design patterns. They dressed a parallel between modularity and abstraction in software systems and modularity and abstraction in patterns. They concluded that when patterns are loosely coupled and abstracted then maintainability, factorability, and reusability are well supported by the patterns. They also concluded on the need for further studies to understand effective pattern constructs and good pattern coupling methods.

Bieman et al. [2, 1] examined common recommended programming styles on several different software systems, with and without patterns, and concluded that in contrast with common claims the use of design patterns can lead to more change prone classes rather than less change prone classes during the evolution of the systems.

3 Problem Formulation

There are little evidence on the impact of design patterns on the quality of software systems. Most of the statements supporting the hypothesis of improvements of the quality are intuitive.

This work aims at quantifying the impact of design patterns on the overall quality of systems. We had the choice between an absolute, a relative, or an empirical quantification.

Due to the lack of a well defined framework for the evaluation of the quality of systems, we chose an empirical quantification consisting of collecting and analyzing evaluations by software developers of certain aspects of the quality of systems that design patterns may impact.

4 Method

We built a questionnaire and carried out a survey electronically during the period of January to May 2007.

4.1 Our Questionnaire

We chose, based on their relevance to design patterns and software systems, the following golden set of quality attributes:

- Related to architecture and design:
  - **Expandability:** The degree to which architectural, data, or procedural design can be extended.
  - **Simplicity:** The degree to which the architecture of the system can be understood without difficulty.
  - **Reusability:** The degree to which a piece of design (or a subset of a piece of design) can be reused in another design.
- Related to implementation:
  - **Learnability:** The degree to which the code source of a system is easy to learn by new developers.
  - **Understandability:** The degree to which the code source of the system can be understood without difficulty.
  - **Modularity:** The degree to which the implementation of functions in a system are independent from one another.
- Related to runtime:
  - **Generality:** The degree to which a system can perform a wide range of functions at runtime.
  - **Modularity at runtime:** The degree to which functions of a system are independent from one another at runtime.
  - **Scalability:** The degree to which the system can cope with large amount of data and computation at runtime.
  - **Robustness:** The degree to which a system continues to function properly under abnormal conditions or circumstances.

Each quality attribute was evaluated using a six-point Likert scale:

```
A - Very positive
B - Positive
C - Not significant
D - Negative
E - Very Negative
F - I don't know
```

For every design pattern in [3] and for every quality attribute from our golden set, the respondents were

asked to assess the impact of the pattern on the quality of a system in which the pattern would have been used appropriately. For example, for the Composite design pattern and Learnability, the respondents were asked to assess the impact of the pattern on the overall Learnability of a system implementing the pattern appropriately.

4.2 Data Collection

The questionnaire was sent to experienced object-oriented developers around the world and posted on three specialized mailing lists, refactoring, patterns-discussion, and gang-of-4-patterns.

We asked the respondents to consider the situation where patterns where used appropriately in programs to solve their corresponding design problems.

Among the answers that we received, we selected the questionnaires of 20 developers with a long experience in the use of design patterns in software development.

Among the selected 20 questionnaires, some respondents did not evaluate the quality of all design patterns. Thus, some patterns have more evaluation than others.

4.3 Data Processing

To answer our question: *is the impact of design patterns on quality attributes positive, neutral, or negative?* and due to the high level of variations between answers, we chose to aggregate answers `A` and `B` and answers `D` and `E`:

```
Positive = A and B
Neutral  = C
Negative = D and E
```

Answers `F` were not considered to assess the impact of design patterns on quality because we considered that the respondent did not know how the pattern impacts the quality attribute.

Using the previous three-point Likert scale, we computed the frequencies of the answers on each quality attribute: Positive, Neutral, and Negative and we carried out a Null test to decide on the impact of the patterns on the quality attributes according to the respondents.

5 Results of the Survey

We present in the following the results for three design patterns: Abstract Factory, Composite, and Flyweight, and the three quality attributes related to design patterns: reusability, expandability, and understandability. Results for all design patterns and quality attributes will be presented in a future work.

Attributes	Positive	Neutral	Negative
Expandability	100.0	0.0	0.0
Simplicity	69.23	15.38	15.38
Generality	76.92	15.38	7.69
Modularity	71.43	21.43	7.14
Modularity at Runtime	53.85	38.46	7.69
Learnability	76.92	7.69	15.38
Understand-ability	69.23	15.38	15.38
Reusability	61.54	23.08	15.38
Scalability	41.67	41.67	16.67
Robustness	8.33	91.67	0.0

Table 1. Impact of Composite on quality attributes.

5.1 Qualitative Analysis

5.1.1 Design Patterns

We chose the following three design patterns to illustrate the opinions of our respondents about the impact of design patterns on quality attributes firstly because of their popularity, they are among commonly used patterns thus we felt that their evaluation would be more accurate, and secondly because they appeared to be considered by our respondents as globally positive (Composite), globally neutral (Abstract Factory), and globally negative (Flyweight). Therefore, we felt that they would be more representative.

Composite. Table 1 presents the evaluations by the respondents of the impact of the Composite pattern on the quality attributes. Looking at the table, it appears that the Composite pattern is mostly perceived as having a positive impact on the quality of systems. All quality attributes are impacted positively but for the scalability and robustness that are not positive. Given the purpose of the Composite pattern, having a neutral impact on scalability is rather surprising.

Abstract Factory. Table 2 presents the evaluations by the respondents of the impact of the Abstract Factory pattern on the quality attributes. The table shows that half the quality attributes is considered as positively impacted while the other half is not. It is not surprising that the pattern is overall judged as neutral given its purpose and complexity. However, it is striking that both learnability and understandability are felt negatively impacted.

Attributes	Positive	Neutral	Negative
Expandability	100.0	0.0	0.0
Simplicity	53.33	13.33	33.33
Generality	78.57	21.43	0.0
Modularity	85.71	7.14	7.14
Modularity at Runtime	46.15	38.46	15.38
Learnability	35.71	28.57	35.71
Understand-ability	38.46	30.77	30.77
Reusability	50.0	42.86	7.14
Scalability	21.43	64.29	14.29
Robustness	0.0	72.73	27.27

Table 2. Impact of Abstract Factory on quality attributes.

Attributes	Positive	Neutral	Negative
Expandability	22.22	44.44	33.33
Simplicity	0.0	22.22	77.78
Generality	11.11	44.44	44.44
Modularity	33.33	33.33	33.33
Modularity at Runtime	11.11	66.67	22.22
Learnability	0.0	20.0	80.0
Understand-ability	0.0	10.0	90.0
Reusability	37.5	12.5	50.0
Scalability	77.78	0.0	22.22
Robustness	22.22	66.67	11.11

Table 3. Impact of Flyweight on quality attributes.

Flyweight. Table 3 presents the evaluations by the respondents of the impact of the Flyweight pattern on the quality attributes. The table reports that this patterns is perceived as impacting negatively all quality attributes but for the scalability. Given the purpose of the pattern, it is not surprising that its impact on scalability is judged positively. The negative perception could be explained by the less frequent use of Flyweight in comparison with Composite and Abstract Factory.

5.1.2 Quality Attributes

We chose the following three quality attributes because it is claimed in [3] that they are improved by the use of design patterns.

Patterns	Positive	Neutral	Negative
A.Factory	46.15	46.15	7.69
Builder	36.36	45.45	18.18
F.Method	60.0	20.0	20.0
Prototype	63.64	0.0	36.36
Singleton	18.18	54.55	27.27
Adapter	66.67	25.0	8.33
Bridge	41.67	16.67	41.67
Composite	58.33	25.0	16.67
Decorator	36.36	18.18	45.45
Facade	36.36	45.45	18.18
Flyweight	37.5	12.5	50.0
Proxy	45.45	36.36	18.18
Ch.Of.Resp	54.55	27.27	18.18
Command	30.0	20.0	50.0
Interpreter	50.0	0.0	50.0
Iterator	72.73	9.09	18.18
Mediator	20.0	50.0	30.0
Memento	28.57	42.86	28.57
Observer	53.85	23.08	23.08
State	20.0	40.0	40.0
Strategy	41.67	33.33	25.0
T.Method	58.33	33.33	8.33
Visitor	28.57	28.57	42.86

Table 4. Impact of design patterns on reusability.

Reusability. Table 4 presents the evaluations by respondents of the impact of design patterns on reusability. Overall, as shown in Table 8, reusability is felt as being slightly more negatively impacted by design patterns, with 12 negative patterns and 11 positive patterns. This is rather surprising as the use of design patterns is claimed to improve reusability according to the GoF.

Expandability. Table 5 presents the evaluations by the respondents of the impact of design patterns on the expendability. All respondents felt that expendability is improved when using design patterns, in conformance with what is expected of using patterns.

Understandability. Table 6 presents the evaluations by the respondents of the impact of design patterns on the understandability. Similarly to reusability, respondents felt that the understandability was rather slightly negatively impacted by the use of patterns.

Patterns	Positive	Neutral	Negative
A.Factory	100.0	0.0	0.0
Builder	90.91	9.09	0.0
F.Method	72.73	9.09	18.18
Prototype	63.64	27.27	9.09
Singleton	9.09	27.27	63.64
Adapter	50.0	41.67	8.33
Bridge	83.33	16.67	0.0
Composite	100.0	0.0	0.0
Decorator	90.91	0.0	9.09
Facade	58.33	16.67	25.0
Flyweight	22.22	44.44	33.33
Proxy	45.45	45.45	9.09
Ch.Of.Resp	91.67	8.33	0.0
Command	66.67	16.67	16.67
Interpreter	63.64	27.27	9.09
Iterator	90.91	9.09	0.0
Mediator	58.33	25.0	16.67
Memento	33.33	55.56	11.11
Observer	85.71	7.14	7.14
State	72.73	18.18	9.09
Strategy	76.92	15.38	7.69
T.Method	84.62	15.38	0.0
Visitor	71.43	7.14	21.43

Table 5. Impact of design patterns on expandability.

Patterns	Positive	Neutral	Negative
A.Factory	38.46	30.77	30.77
Builder	81.82	9.09	9.09
F.Method	45.45	27.27	27.27
Prototype	58.33	16.67	25.0
Singleton	91.67	8.33	0.0
Adapter	50.0	25.0	25.0
Bridge	50.0	33.33	16.67
Composite	75.0	16.67	8.33
Decorator	45.45	9.09	45.45
Facade	81.82	18.18	0.0
Flyweight	0.0	10.0	90.0
Proxy	33.33	50.0	16.67
Ch.Of.Resp	33.33	33.33	33.33
Command	33.33	33.33	33.33
Interpreter	63.64	0.0	36.36
Iterator	50.0	41.67	8.33
Mediator	58.33	25.0	16.67
Memento	33.33	55.56	11.11
Observer	42.86	35.71	21.43
State	54.55	0.0	45.45
Strategy	69.23	23.08	7.69
T.Method	38.46	38.46	23.08
Visitor	21.43	21.43	57.14

Table 6. Impact of design patterns on understandability.

5.2 Quantitative Analysis

Using the results obtained by aggregating the previous data in Tables 1, 2, 3, 4, 5, and 6, and we carried out a Null hypothesis test to quantify the impact of the design patterns on the quality attributes. We use the frequencies of Positive and non-positive (combined Neutral and Negative answers) to decide on the impact of a given pattern on a specific quality attribute.

For a given question about the impact of a pattern on a quality attribute, we considered the random variable X, that takes the value 0 when the impact of the pattern on the attribute is positive and 1 when the impact is not positive. We defined P as the probability that the pattern does not impact positively the attribute. The probability that the pattern impacts positively the attribute is therefore $1-P$. Considering the N respondents $j = 1, ..., N$ answering the question, we viewed their answers as occurrences of the random variable X and noted them: $X_1, X_2, ...X_N$. Then, we set our Null hypothesis to be H_0: The impact of the pattern on the quality attribute is positive, which yields, in terms of probability, to $P \leq \frac{1}{2}$. The alternative hypothesis is then H_1: The pattern does not impact positively the attribute, i.e., $P > \frac{1}{2}$. Hence, our decision rule is:

- We confirm H_0 if f_N is not high enough;
- We confirm H_1 if f_N is high enough.

where f_N is the frequency of the respondents who answered that the pattern impacts negatively or does not impact the attribute.

The risk we encountered by rejecting the Null hypothesis H_0, i.e, the pattern positively impacts the quality attribute, is then: $1-F(f_N)$, where F is the cumulative density of the Bernoulli distribution $\beta(N, \frac{1}{2})$.

The Null hypothesis test yields the results summarized in Tables 7 and 8. The analysis of the results of our survey revealed that in contrary to what is commonly admitted in the literature (which is that the use of design patterns yields to architectures that are reusable, simple and more understandable), the reality of the use of patterns is different. Developers consider that although patterns are useful to solve design problems, they do not always improve the quality of systems

in which they are used. Some patterns like Flyweight are even considered bad for the quality of systems. A large number of respondents consider that they sensibly decrease simplicity, learnability, and understandability.

6 Conclusion

In this paper, we presented a study of the impact of design patterns on quality attributes. This empirical study was performed by asking respondents their evaluations of the impact of all design patterns on several quality attributes. We concluded that, contrary to popular beliefs, design patterns do not always improve reusability and understandability, but that they do improve expendability.

However this study stands only on the opinion of a surveyed population of experienced developers and we cannot consider its results as free from uncertainty. In particular, some design patterns received no evaluations from some respondents because they are less known and used or, possibly, because they are judged up-front as impacting negatively quality as one respondent suggested. Moreover, the 20 respondents may not be representative of the general population of software developers.

In future work, we plan to carry out a wider survey by detailing our questionnaire and broadcasting it to more respondents. The questionnaire is available on the Internet at `http://www.iro.umontreal.ca/~ptidej/Questionnaire.pdf` or `http://ptidej.dyndns.org/downloads/` (it may take some minutes to load as it weighs 4 MB). We are looking forward receiving your kind contributions.

Acknowledgments

Foutse Khomh and Yann-Gaël Guéhéneuc were partially supported by NSERC, Discovery Grant.

We would also like to thank all the respondents of our questionnaire for their evaluations of the patterns and for all their very interesting comments.

References

[1] J. Bieman, G. Straw, H. Wang, P. W. Munger, and R. T. Alexander. Design patterns and change proneness: An examination of five evolving systems. In *Proceedings of the* 9^{th} *international Software Metrics Symposium*, pages 40–49. IEEE Computer Society Press, September 2003.

[2] J. M. Bieman, R. Alexander, P. W. M. III, and E. Meunier. Software design quality: Style and substance. In *Proceedings of the* 4^{th} *Workshop on Software Quality*. ACM Press, March 2001.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1^{st} edition, 1994.

[4] W. B. McNatt and J. M. Bieman. Coupling of design patterns: Common practices and their benefits. In *Proceedings of the* 25^{th} *Computer Software and Applications Conference*, pages 574–579. IEEE Computer Society Press, October 2001.

[5] L. Tahvildari and K. Kontogiannis. On the role of design patterns in quality-driven re-engineering. In *Proceedings of the*6^{th} *European Conference on Software Maintenance and Reengineering*, pages 230–240. IEEE Computer Society, March 2002.

[6] B. Wydaeghe, K. Verschaeve, B. Michiels, B. V. Damme, E. Arckens, and V. Jonckers. Building an OMT-editor using design patterns: An experience report. http://info.vub.ac.be/bwydaegh/tekst/paers/tools98/html/tools98.html, 1998.

Attributes	Composite		A.Factory		Flyweight	
	E	R(%)	E	R(%)	E	R(%)
Expendability	+	0.0	+	0.0	-	1.76
Simplicity	+	5.92	+	30.36	-	0.0
Generality	+	1.76	+	1.76	-	0.15
Modularity	+	5.92	+	0.37	-	5.92
Modularity at Runtime	+	30.36	-	30.36	-	0.15
Learnability	+	1.76	-	15.09	-	0.0
Understandability	+	5.92	-	15.09	-	0.0
Reusability	+	15.09	+	50.0	-	15.09
Scalability	-	30.36	-	1.76	+	1.76
Robustness	-	0.15	-	0.0	-	1.76
	8 + / 2 -		5 + / 5 -		1 + / 9 -	

Table 7. Estimation of the impact of the three design patterns on quality attributes.

Design Patterns	Expendability(%)		Understandability(%)		Reusability(%)	
	E	R(%)	E	R(%)	E	R(%)
A.Factory	+	0.0	-	15.09	+	50.0
Builder	+	0.15	+	0.37	-	15.09
F.Method	+	1.76	-	30.36	+	15.09
Prototype	+	30.36	+	30.36	+	30.36
Singleton	-	0.15	+	0.15	-	0.37
Adapter	+	30.36	-	30.36	+	5.92
Bridge	+	0.37	+	50.0	-	30.36
Composite	+	0.0	+	5.92	+	15.09
Decorator	+	0.15	-	30.36	-	5.92
Facade	+	30.36	+	1.76	-	5.92
Flyweight	-	1.76	-	0.0	-	15.09
Proxy	-	30.36	-	5.92	+	50.0
Ch.Of.Resp	+	0.15	-	5.92	+	30.36
Command	+	5.92	-	5.92	-	5.92
Interpreter	+	5.92	+	5.92	+	30.36
Iterator	+	0.15	+	50.0	+	5.92
Mediator	+	30.36	+	30.36	-	1.76
Memento	-	5.92	-	30.36	-	15.09
Observer	+	0.15	-	30.36	+	50.0
State	+	5.92	+	30.36	-	1.76
Strategy	+	1.76	+	15.09	-	30.36
T.Method	+	0.37	-	15.09	+	30.36
Visitor	+	5.92	-	1.76	-	1.76
	19 + / 4 -		11 + / 12 -		11 + / 12 -	

Table 8. Estimation of the impact of design patterns on the three quality attributes